Part THREE
Lambda Expressions

Chapter EIGHT
Functional Interfaces

Exam Objectives

Create and use Lambda expressions.

Simplifying things

Suppose we have a program that has a list of cars and we need to search for all compact cars.

We can have something like this to do the work:

```
List<Car> findCompactCars(List<Car> cars) {
     List<Car> compactCars = new ArrayList<Car>();
     for(Car car : cars) {
         if(car.getType().equals(CarTypes.COMPACT)) {
             compactCars.add(car);
         }
     }
     return compactCars;
}
```

Easy. But the next day, the users realize they also need to search for cars that cost more than 20,000 USD.

So we come up with something like this:

```
List<Car> findCompactCars(List<Car> cars) {
     List<Car> twentyKCars = new ArrayList<Car>();
     for(Car car : cars) {
         if(car.getCostUSD() > 20000) {
             twentyKCars.add(car);
         }
     }
     return twentyKCars;
}
```

Now look at the code. It's practically the same. The only differences are the lines:

```
car.getType().equals(CarTypes.COMPACT)
```

And:

```
car.getCostUSD() > 20000
```

What if we need to filter by another condition in the future?

It's wrong to have duplicate or copy-pasted code, it's not very flexible to change and error-prone.

The Object-Oriented Approach

Since Java is an object-oriented let's leverage this by solving the problem using a popular design pattern, Strategy.

This pattern does just what we need, encapsulate the behavior that varies (algorithm) and makes all these behaviors interchangeable.

The recommended way to implement the Strategy pattern in Java is with an interface so we can create implementations with for each algorithm (in our case, the conditions to test)

So let's code an interface that can contain the search condition that varies, for example:

```
interface Searchable {
     boolean test(Car car);
}
```

And place the code that varies into implementations of that interface:

```
class CompactCarSearch implements Searchable {
     public boolean test(Car car) {
         return car.getType().equals(CarTypes.COMPACT);
     }
}
```

And:

```
class TwentyKCarSearch implements Searchable {
     public boolean test(Car car) {
         return car.getCostUSD() > 20000;
     }
}
```

That way, we can make the method that does the actual search a little more general:

```
List<Car> findCars(List<Car> cars, Searchable s) {
     List<Car> searchedCars = new ArrayList<Car>();
     for(Car car : cars) {
         if(s.test(car)) {
             searchedCars.add(car);
         }
     }
```

```
    return searchedCars;
}
```

And call the function in this way:

```
List<Car> compactCars = findCars(cars, new CompactCarSearch());
List<Car> twentyKCars = findCars(cars, new TwentyKCarSearch());
```

We don't have duplicate code anymore.

If the user wants another way to search for a car, instead of copy-pasting a search method, now we just implement another class with a condition.

We are now more flexible to change.

But it's still a **COMPLEX** solution. Instead of having two methods we have two classes and one interface.

What if we turn the implementations into anonymous classes? For example:

```
List<Car> compactCars = findCars(cars,new Searchable() {
    public boolean test(Car car) {
        return car.getType().equals(CarTypes.COMPACT);
    }
});
```

A little bit better, we can even assign the class to a variable if we want to use it in other parts of our program:

```
Searchable compactCarSearch = new Searchable() {
    public boolean test(Car car) {
        return car.getType().equals(CarTypes.COMPACT);
    }
});
List<Car> compactCars = findCars(cars, compactCarSearch);
```

Not bad, but wouldn't it be great to just pass the condition to the method? Like this for example:

```
// Don't even try, it won't work
List<Car> compactCars = findCars(cars,
            car.getType().equals(CarTypes.COMPACT));
```

A new type of value

Basically, in Java we work with two types of value, primitive values, and references to objects.

Both can be used as the arguments of a method:

```
method1(3);
method2(new Object());
```

So if we want to pass some piece of code to a method, we have to wrap it in an object (like in the previous example).

But in Java 8, we can pass that piece of code directly through the use of a lambda expression.

Coincidentally, to use a lambda expression instead of an object in the previous example, we have to start from the same interface:

```
interface Searchable {
     boolean test(Car car);
}
```

A functional interface.

A functional interface

The starting point to learn about lambda expressions is learning about functional interfaces.

A functional interface is any interface that has exactly **ONE ABSTRACT** method.

This is a tricky definition. Many people think that just having one method makes an interface functional (since interface methods are abstract by default), but it doesn't.

Take for example default methods. Since default methods have an implementation, they are not abstract. So interfaces like the following, are considered functional:

```
interface A {
     default int defaultMethod() {
         return 0;
     }
     void method();
}
interface B {
     default int defaultMethod() {
         return 0;
     }
     default int anotherDefaultMethod() {
         return 0;
     }
     void method();
}
```

If an interface declares an abstract method with the signature of one of the methods of `java.lang.Object` , it doesn't count toward the functional interface method count since any implementation of the interface will have an implementation of the method (since all classes extend from `java.lang.Object`).

So an interface like the following is considered functional:

```
interface A {
     boolean equals(Object o);
     int hashCode();
     String toString();
     void method();
}
```

A more confusing scenario is when an interface inherits a method that is equivalent but not identical to another:

```
interface A {
     void method(List<Double> l);
}
interface B extends A {
```

```
     void method(List l);
}
```

In this case, the method is the same, so it's taken as one method. The class that implements `B` will have to implement the method that can legally override all the abstract methods:

```
void method(List l);
```

By the way, neither is an empty interface considered functional.

The key here is to have **EXACTLY ONE ABSTRACT** method; that's why these interfaces are also called Single Abstract Method interfaces (SAM Interfaces).

To make things easier, Java 8 also introduced the `@FunctionalInterface` annotation, which generates a compile-time error when the interface you have annotated is not a valid functional interface.

```
// This won't compile
@FunctionalInterface interface A {
     void m(int i);
     void m(long l);
}
```

Java interfaces that only have one method declared in their definition are now annotated as functional interfaces. Some examples are:

- `java.lang.Runnable`
- `java.util.Comparator`
- `java.util.concurrent.Callable`
- `java.awt.event.ActionListener`

But remember, this annotation is just to help you; having it won't make an interface functional.

In the next chapter, we will see how functional interfaces and lambda expressions are related.

Key Points

- A functional interface is any interface that has exactly one abstract method.
- Since default methods have an implementation, they are not abstract so a functional interface can have any number of them.
- If an interface declares an abstract method with the signature of one of the methods of `java.lang.Object`, it doesn't count toward the functional interface method count.
- A functional interface is valid when it inherits a method that is *equivalent* but not *identical* to another.
- An empty interface is not considered a functional interface.
- A functional interface is valid even if the `@FunctionalInterface` annotation would be omitted.
- Functional interfaces are the basis of lambda expressions.

Self Test

1. Given:

```
interface A {
     default int m1() {
         return 0;
     }
}
@FunctionalInterface
public interface B extends A {
     static String m() {
         return "static";
     }
}
```

Which of the following statements are true?
A. Compilation fails
B. It compiles successfully
C. It compiles only if interface `B` declares a default method
D. An exception occurs at runtime if this interface is implemented by a class

2. Given:

```
@FunctionalInterface interface C {
     int m(int i);
     long m(long i);
}
```

Which of the following statements are true?
A. This code compiles successfully
B. If we remove the annotation, this code will compile
C. If we remove one method, this code will compile
D. The `@FunctionalInterface` annotation makes this interface functional

3. Given:

```
public interface D {
     int sum(int a, int b);
     default int subtract(int a, int b) {
         return a - b;
     }
}
```

Which of the following statements are true?
A. This code compiles successfully
B. This code doesn't compile
C. This is an example of a functional interface
D. Removing the sum method would make this interface functional

4. Which of the following interfaces of the Java API can be considered functional?
A. `java.util.concurrent.Callable`
B. `java.util.Map`
C. `java.util.Iterator`
D. `java.lang.Comparable`

5. Given:

```
public interface E {
     static int sum(int a, int b) {
         return a + b;
     }
}
```

Which of the following statements are true?
A. This code doesn't compile
B. This code compiles successfully
C. This is an example of a functional interface
D. Adding the `@FunctionalInterface` annotation would make this interface functional

Open answers page

---

07. Collections 09. Lambda Expressions